Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 128 (2005) 279–294
www.elsevier.com/locate/entcs

Efficient Model Checking of Hardware Using Conditioned Slicing

Shobha Vasudevan[1]

Computer Engineering Research Center
University of Texas at Austin

E. Allen Emerson[2]

Department of Computer Sciences
University of Texas at Austin

Jacob A. Abraham[3]

Computer Engineering Research Center
University of Texas at Austin

Abstract

In this work, we present an abstraction based property verification technique for hardware using *conditioned slicing*. We handle safety property specifications of the form $G(antecedent \Longrightarrow consequent)$. We use the antecedent of the properties to create our abstractions, *Antecedent Conditioned Slices*. We extend conditioned slicing to Hardware Description Languages (HDLs). We provide a theoretical foundation for our conditioned slicing based verification technique. We also present experimental results on the Verilog RTL implementation of the USB 2.0. We demonstrate very high performance gains achieved by our technique when compared to static program slicing, using state-of-the-art model checkers.

Keywords: Hardware Verification, Program Slicing, Conditioned Slicing, LTL property, Abstraction based Verification, Safety properties, Model Checking, Hardware Description Languages

[1] Email: shobha@cerc.utexas.edu
[2] Email: emerson@cs.utexas.edu This research is supported in part by NSF grants CCR-009-8141 & ITR-CCR-020-5483, and SRC Contract No. 2002-TJ-1026.
[3] Email: jaa@cerc.utexas.edu

doi:10.1016/j.entcs.2005.04.017

1 Introduction

State space reduction techniques have been extensively explored to make the model checking [5] problem tractable. Among these, abstraction techniques are emerging as the key to verification of large systems, especially those that are not finite state.

In the context of hardware verification, a majority of the state space reduction techniques have been applied at the gate (boolean netlist) level descriptions of the designs. It is desirable, however, to apply these techniques at the Register Transfer Level (RTL) of these designs, described in a Hardware Description Language (HDL). Source-to-source transformations of the HDL descriptions of designs can greatly simplify the synthesis of large designs, and decrease the verification complexity. Since RTL descriptions of designs can be likened to "programs" in software, manipulation of the RTL code allows for using parallel techniques in software for program size reduction. However, the computational paradigm involves concurrency and non determinism, thereby making them fundamentally differ from sequential programming languages.

Program Slicing, introduced by Weiser [24],[23] is an abstraction technique used to statically analyze programs and decompose them into parts that faithfully represent the original program within a behavioral domain. Program slicing has been applied to various software engineering disciplines where manipulation of large programs, and hence their decomposition is desirable. In the past, program slicing has been extended to HDLs [6], [11], [20]. Program slicing has also been successfully applied to hardware verification [6], [21].

Weiser's slicing has also been called static slicing. A slicing technique that improves over static program slicing is *conditioned slicing* [1], [3]. Conditioned slicing augments static program slicing by specifying some initial states of interest in the slicing criterion. Conditioned slicing has been shown to produce smaller and more meaningful abstractions than static slicing.

In this paper, we introduce an abstraction technique for property based hardware verification using conditioned slicing. Conditioned slicing has been used for program comprehension [15], reuse [3], migration [2] and re-engineering [4]. However, to the best of our knowledge, this is the first application of conditioned slicing to verification.

Conditioned slicing has been defined for sequential programs. We extend conditioned slicing to HDLs. We develop a technique for computing conditioned slices of HDLs from the antecedent of property specifications of the form $G(antecedent \implies consequent)$. In this paper, we handle safety properties of the above form. Since safety properties constitute the majority of properties required to verify systems, our technique is of high practical importance.

We present *Antecedent Conditioned Slices*, the abstractions created by our technique. We argue that our technique is effective for hardware verification and provide a theoretical basis for it. We also provide experimental results to show the substantial performance gains of using this technique, as compared to state-of-the-art model checking techniques. Experiments are shown on an RTL implementation of the USB 2.0 protocol.

The outline of the paper is as follows. Section 2 gives a background of program slicing techniques and presents some fundamental definitions. Section 3 introduces conditioned slicing, while Section 4 gives the details of computing conditioned slices using dependence graphs. Section 5 describes the application of conditioned slicing to HDLs with an example. In Section 6, our framework for applying conditioned slicing to hardware verification is described and shown to be correct. Experimental results are presented in Section 7. Section 8 discusses the merits and demerits of the presented technique. We conclude with Section 9.

2 Slicing Techniques

Slicing, in the most general sense, is a program transformation involving statement deletion, that preserves some projection of the semantics of the original program.

The aspect of the program that must be preserved is application specific, and is captured by the slicing criterion. We present here, some necessary background for program slicing.

Definition 2.1 Static slicing criterion
A slicing criterion of a program P with an input alphabet Σ, is a pair $\langle i, V \rangle$ such that i is a statement in P and $V \subseteq \Sigma$.

A set of statements I_s is said to *affect* the values of V at i in a given slicing criterion $\langle i, V \rangle$, if I_s defines (writes into) a subset of V that is used in (read by) i.

Definition 2.2 Static slice for programs
A slice S of a program P on a slicing criterion $\langle i, V \rangle$ is a subset of the statements of P that affect the values of V at i.

Program slicing has been used for a variety of tasks, including debugging, maintenance and testing. A detailed survey of program slicing techniques can be found in [19]. A static slice preserves the projection of the semantics of the original program for every possible execution of the program. This can result in very large slices [15], [12]. Many slicing algorithms have been proposed

as variations of static slicing, to create smaller slices. Dynamic slicing [12] makes assumptions about a specific input (test case) to a program. Hybrid approaches [18], [22] combine both static and dynamic slicing.

3 Conditioned Slicing

Canfora et al. [1] introduced the notion of *conditioned slicing*, that forms a theoretical bridge between static and dynamic slicing. Conditioned slices are constructed with respect to a set of possible input states, characterized by a first order predicate logic formula. Conditioned Slicing augments static slicing by introducing a condition that specifies the initial set of states in the criterion. It does not give specific inputs, unlike dynamic slicing. This slicing technique, therefore allows slicing with respect to the initial states of interest, or initial constraints in the program. We present some basic definitions of conditioned slicing that appear in the literature.

Definition 3.1 Conditioned Slicing criterion
Let Σ be the set of input variables to the program P. Let C be a first order predicate logic formula on the variables in Σ. A conditioned slicing criterion is a triple $\langle C, i, V \rangle$, where i is a statement in the program, and $V \subseteq \Sigma$.

Definition 3.2 Conditioned Slicing for programs
A conditioned slice of a program P on a conditioned slicing criterion $\langle C, i, V \rangle$ consists of all the statements and predicates of P that affect the values of the variables in V at i, when the condition C holds true.

Tip [18] introduced a more restricted form of conditioned slicing called constraint based slicing. In all these cases, the *condition* that specifies the set of initial states, and is used for slicing is a first order predicate logic formula. We will refer to this condition as the *conditional predicate*, or simply *predicate*. We will use the term *conditioning* to mean the process of obtaining a conditioned slice with respect to a given conditional predicate C.

Conditioned slicing is a significant improvement over static, dynamic or quasi-static [1] slicing, since it subsumes all of these as special cases.

In situations where the initial set of constraints for the program analysis are known, this technique can be employed to get much smaller slices than those produced by static slicing. This technique can, therefore be used to simplify the code, before applying a traditional static slicing algorithm.

4 Dependence graph based slicing

Ottenstein & Ottenstein define slicing as a reachability problem in a dependence graph representation of the program [17].

They use Program Dependence Graphs (PDGs) [9] for static slicing of single procedure programs. The statements and predicates of a program correspond to the vertices of the PDG and the edges correspond to data and control dependences between statements. In dependence graph based slicing approaches, the slicing criterion is identified with a vertex v in the PDG. The i in the slicing criterion $\langle i, V \rangle$ corresponds to v in the PDG, while V stands for the set of all variables defined or used at v.

For slicing of multi-procedure programs, System Dependence Graphs (SDGS) were introduced [10].

An SDG combines the *procedure dependence graphs* of all the called procedures of a program, alongwith the *program dependence graph* of the main program by allowing edges that can model procedure calls.

Let the program dependence graph for a program P, denoted by G_P be a directed graph connected by several kinds of edges. The vertices $v_1, v_2 \ldots v_n$ of G_P represent the assignment statements and control predicates that occur in P. The edges represent dependences among program components. An edge can be of one of the following broad types.

Definition 4.1 Control Dependence Edge
A control dependence edge from v_1 to v_2, where v_1 is a predicate vertex, implies that the truth of the predicate expression represented by v_1 determines whether or not v_2 is executed.

Definition 4.2 Flow Dependence Edge
A flow dependence edge from v_1 to v_2, implies that there is some variable x, that is defined at v_1 and used at v_2 and there is an execution path from v_1 to v_2 along which, there is no assignment to x.

4.1 Conditioned Slicing using PDGs

We present here conditioned slicing using a dependence graph approach. To the best of our knowledge, such a treatment of conditioned slicing is novel.

Figure 1 shows an example program written in pseudo code. The PDG for the program is shown in Figure 2. In order to find a static slice for the program with respect to slicing criterion $\langle 11, B \rangle$, we find all the reaching definitions of B at node 11. Then, we find the set of all reachable nodes from these nodes in the PDG. This set $\{1, 2, 3, 4, 6, 7, 9\}$, gives us the desired static slice. The nodes are shown in bold in the figure.

```
        begin
1:              read(N);
2:              A = 1;

3:              if (N < 0) {
4:                  B = f(A);
5:                  C = g(A);
6:              } else if (N > 0) {
7:                  B = f'(A);
8:                  C = g'(A);
                } else {
9:                  B = f''(A);
10:                 C = g''(A);
                }
11:             print(B);
12:             print(C);
         end
```

Fig. 1. Example Program written in pseudo-code

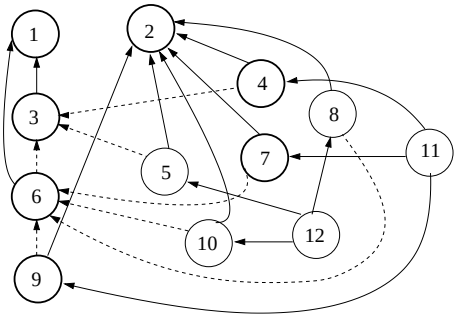

Fig. 2. Program Dependence Graph of the program. The solid edges denote data dependency and the dashed edges denote control dependencies. The vertices in bold denote the Static slice of the program with respect to the variable B at statement 11.

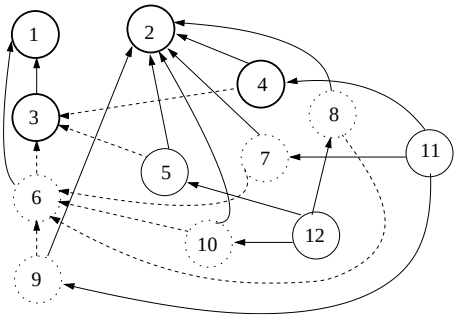

Fig. 3. Conditioned Slice with respect to the predicate $N < 0$. A vertex is made solid if it is ever executed and made bold if it gets traversed while computing the slice. The dotted vertices are not executed when the predicate is true

Now, consider the conditioned slicing of the program with respect to the slicing criterion $\langle C, 11, B\rangle$, where C corresponds to the predicate $(N < 0)$.

To obtain the conditioned slice for a given predicate C, we *project* the PDG

with respect to the predicate, and then use the static slicing algorithm on the projected PDG. Figure 3 shows the application of this technique to obtain the conditioned slice for the mentioned criterion. Initially, all the vertices in the graph are drawn dotted. All the statements that would get executed when the predicate C is satisfied, are marked, and the corresponding vertices in the graph are made solid. The graph is then traversed only for solid (marked) vertices.

The set of all vertices reached during the traversal are made bold. This set $\{1, 2, 3, 4\}$ gives the desired conditioned slice. The conditioned slice, therefore, contains only those statements that get executed when C evaluates to *true*. It is evident from this example that the conditioned slice is typically much smaller than the static slice.

5 Conditioned Slicing for HDLs

We present here, an extension of conditioned slicing to HDLs. The HDL computational paradigm differs fundamentally from traditional languages, since HDLs model non-halting, reactive systems with concurrently running processes. The processes are not explicitly called, as they are in a program's procedures. Instead, they communicate through signal dependency [6]. In order to extend conditioned slicing to HDLs, we use a few definitions from earlier work in program slicing [24] and its extension to HDLs [6], [20] with minor modifications. We also use a few definitions from earlier work in conditioned slicing [3], [1].

Let M be a Verilog program [1] $\|_{i=1}^{k} P_i$ with k concurrent processes P_i, where $\|$ is the parallel composition operator [20].

Definition 5.1 Signal dependence
A process region P is said to be signal dependent on a statement i, if i assigns a value to a signal which is present in the sensitivity list of P.

Definition 5.2 Inter process CFG
An inter-process control flow graph G for a Verilog program is the structure $\langle G_1, G_2 \ldots G_k, Esd \rangle$ where $G_1, G_2 \ldots G_k$ are the control flow graphs representing the processes in the program, and E_{sd} is the set of edges representing the signal dependencies between the processes.

Definition 5.3 Static Slicing for HDLs
An inter-process slice S_p within a Verilog program M, on a given criterion $\langle i, V \rangle$ is an executable subset of M obtained recursively containing all the

[1] The term "program" is used with the same meaning as in [6] for VHDL designs. A Verilog program is a set of concurrent processes that executes as a series of simulation cycles.

statements that affect the values of V at i within the process P in which i is defined, and all the slices on the slice criterion $\langle i_s, V_s \rangle$ where i_s is the set of statements defining the set of signals V_s on which process P is dependent.

Program Slicing of HDLs has been modeled as a dependence graph node reachability problem in [6]. The CFGs constructed for every process also model the non-halting reactive nature of digital hardware. Each concurrent process has a corresponding Process Description Graph. The PDG for each process is modified for HDLs by making provision for inter-process communication through signals. Implicit procedure calls are generated in the inter-process Control Flow Graph, whenever a signal is potentially assigned. The PDGs of the processes are connected appropriately to form an SDG. The slices are computed by following the chains of dependences represented in the edges of the SDG. We now extend the above technique to incorporate conditioned slicing of HDLs.

Definition 5.4 Conditioned process
A conditioned process $P(C)$, for a process P and a condition C, is an entity containing the set of statements of P that can be executed when C holds true.

To compute $P(C)$, the predicates at all nodes that are on an execution path of G_P are analyzed.

If a predicate p that lies on an execution path is such that $C \Longrightarrow p$, only the path generated by the *true* labelled branch is included. If a predicate p that lies on an execution path is such that $C \Longrightarrow \bar{p}$, only the path generated by the *false* labelled branch is included. If the condition C does not bind p to any truth value, both *true* and *false* paths are included.

Definition 5.5 Conditioned Program
A conditioned program with respect to the condition C is represented as $M(C) \equiv \|_{i=1}^{k} P(C)_i$ where $P(C)_i$ is a conditioned process.

Definition 5.6 Inter-process Conditioned Slicing
An inter-process conditioned slice S_c for a Verilog program M, on a given criterion $\langle C, i, V \rangle$ is a subset of M obtained recursively containing

a) all statements that affect the values of V at i within the process P where i is defined, when the condition C holds true
b) all the conditioned slices on the slice criterion $\langle C, i_s, V_s \rangle$, where i_s is the set of statements that define the set of signals V_s on which P is signal dependent, when the condition C holds true.

The conditioned slice of a Verilog program M can be computed using its SDG representation. The predicate C in the slicing criterion is applied to

all the PDGs of all the processes P_i in M. The resulting conditioned process $P(C)_i$ is marked in its PDG. The conditioned program $M(C)$, is thus obtained from all the conditioned processes. $M(C)$ is now marked on the SDG. The slices are computed by finding the transitive closure of the control flow and signal dependencies of the conditioned program, $M(C)$ in the SDG.

```
        P1                          P2                  P3
always @(clk or valid)     always @(clk)       always @(clk or flag)
begin                      begin               begin
   if (valid)                 valid = rst;        start = flag;
      result = a+b;           if (valid)       end
   else                          flag = 1;
      result = a-b             else
                                  flag = 0;
end                        end
```

Fig. 4. Example Verilog program. The three "always" blocks represent concurrent processes.

Figure 4 illustrates an example program of a hardware system, in Verilog HDL. The three processes `P1,P2` and `P3` correspond to the concurrently executing `always` blocks in Verilog. The parentheses of the always blocks list the signals on which each process is dependent. Now, consider the slicing criterion $\langle C, end_{P1}, result \rangle$, where C corresponds to the predicate $valid = true$ and end_{P1} corresponds to the *end* statement of process $P1$. We apply this predicate on each of the three processes, to obtain the corresponding conditioned processes, $P(C)_1$, $P(C)_2$ and $P(C)_3$. The resulting conditioned program is shown in Figure 5. The portions of the code which can be executed, when the predicate *valid* is true, are shown. The conditioned program is now analyzed for determining the slice with respect to the given criterion. The transitive closure of the data, control and signal dependencies of the relevant variables yields the program of Figure 6. It can be seen that `P1` is included in the slice due to the signal dependence of `P1` on `P2` with respect to the variable *valid*. `P3` is eliminated, since the variables in the slicing criterion do not have any dependencies on it.

```
        P1                          P2                  P3
always @(clk or valid)     always @(clk)       always @(clk or flag)
begin                      begin               begin
   if (valid)                 valid = rst;        start = flag;
      result = a+b;           if (valid)       end
end                              flag = 1;
                           end
```

Fig. 5. The conditioned program, for the predicate (valid = true). Each process is a conditioned process.

```
        P1                          P2
always @(clk or valid)      always @(clk)
begin                       begin
   if (valid)                  valid = rst;
      result = a+b          end
end
```

Fig. 6. The slice obtained by statically slicing the conditioned program.

6 Verification using Conditioned Slicing

6.1 Antecedent Conditioned Slices

We use conditioned slicing for verification of hardware designs described in Verilog HDL. Our technique aims at reducing the state space of the design, by slicing away the part of the design irrelevant to the property being verified. We assume that the properties are specified as LTL [16], [14] formulae of the form, $G(antecedent \implies consequent)$. For these properties, we can use the antecedent to specify the set of initial states that we are interested in. *The antecedent, therefore, forms the condition in the slicing criterion.* All the statements that would get executed when the antecedent is true (or the condition is satisfied) are retained in the slice. The statements on the paths that cannot get executed when the antecedent is false, are removed. The reduced program still preserves its behavior *with respect to the property being checked.* We therefore create property preserving abstractions using conditioned slicing.

All prior art in verification using program slicing uses static program slicing techniques. While slicing property specifications of the form $antecedent \implies consequent$, these techniques retain the set of all statements of the program where the antecedent is true, *as well as those where it is not.* This is because static slicing retains all possible executions of the relevant variables.

However, in property based verification, *we do not need to check the states where the antecedent is false.* In these cases, static slices might be too large and include statements that are not of interest. We introduce a precise abstraction on the basis of conditioned slicing, *Antecedent Conditioned Slices.*

Definition 6.1 Antecedent Conditioned Slice
Let h be an LTL formula $G(p \implies q)$. Let P' be the conditioned slice of a program P with respect to the slicing criterion $\langle p, i, V \rangle$. The Antecedent Conditioned Slice S_p with respect to the slicing criterion $\langle p, i, V \rangle$ is a subset of P' such that $G(p \implies G(p))$ in the scope of S_p.

Antecedent Conditioned Slices, therefore, contain all and only those statements where the condition remains true through the scope of the slice. These are significantly smaller slices, that can reduce verification state spaces significantly. We provide a theoretical justification for verification using Antecedent

Conditioned Slices below.

6.2 Justification

Let us consider a program P. Let Σ be the set of input variables to the program P. Let a be predicate, such that $a \in \Sigma$. Let $M = (S, R, I)$ be the Kripke structure representing program P, where S and R represent the states and the transitions and I represents the set of initial states of the program. Consider an LTL or bounded LTL formula, h, of the form $G(p \Rightarrow q)$ or of the form $G^{\leq k}(p \Rightarrow q)$, where p and q are the antecedent and the consequent respectively. Let h be used to describe a property to be verified in P. Now, if $p \equiv a$, or the antecedent corresponds to the truth value of predicate a, we can use p as the condition in the conditioned slicing criterion. $M \models h$ implies that for all paths x in M starting at a state S_0, where $S_0 \in I$, we have $M, x \models h$.

We will now slice P with respect to the conditioned slicing criterion $\langle p, i, V\rangle$, which is obviously equivalent to the criterion $\langle a, i, V\rangle$.

Let $P_{|a}$ be the Antecedent Conditioned Slice obtained from P with respect to the criterion $\langle a, i, V\rangle$. From Definition 6.1, $P_{|a}$ comprises only those sets of states of P, where the antecedent a is true.

Let $M_{|a}$ be the substructure obtained by restricting the states in M to those satisfying the antecedent a. Let $N = (S', R', I')$ be the Kripke structure representing the Antecedent Conditioned Slice $P_{|a}$, such that S' and R' represent the states and the transitions and $I' = I$. It is obvious that, by construction, $M_{|a}$ is a substructure of N and N is a substructure of M.

In order to prove the correctness of the Antecedent Conditioned Slice obtained, we need to prove that the property h holds on the original program if and only if holds on the Antecedent Conditioned Slice. We present here, a proof outline for this theorem. We detail the proof for bounded LTL formulae.

Theorem 6.2 *$M \models h \iff N \models h$ where $h = G^{\leq k}(a \Rightarrow q)$.*

Proof We say a state $T \in M$ is "close" if there a path $x \in M$ from some initial state $s_0 \in I$ to T of length at most k. The proof follows from proving Lemma 1 and Lemma 2.

Lemma 1 *Let $M \models h$. Then, $N \models h$.*

Proof:

All "close" states in M satisfy $a \Longrightarrow q$. Since N is a substructure of M, this is also true of all the close states in N.

Lemma 2 *Let $N \models h$. Then, $M \models h$.*

Proof: All close states in N satisfy $a \Longrightarrow c$. These states include all the close

states of $M_{|a}$. Thus all close states of M that satisfy a must also satisfy $a \Longrightarrow q$. All states of M that satisfy $\neg(a)$, including the close states, satisfy $a \Longrightarrow q$ vacuously. Therefore, we infer that all close states of M satisfy $a \Longrightarrow c$.

7 Experimental Results

We provide experimental results on the Verilog RTL implementation of the USB 2.0 Function Core. The USB is a standard interconnect between computers and peripherals. This core operates at full and high speed rates (12 and 480 Mb/s). The source code can be found at [7]. The properties chosen for verification were from the USB 2.0 core specification document [8]. These properties were involved with the many state machines in the implementation, and were essentially control based properties. The properties have been listed below as LTL formulae of the form $G^{\leq k}(p \Longrightarrow q)$, where k is the bound (in this case $k = 24$) and also explained in English, for the sake of readability. The variables used in the LTL formulae are the signal names in the Verilog code. The Verilog state machines that correspond to the given property are given in parentheses.

- **P1**: $G^{\leq k}((state = IDLE) \wedge (ep\_stall) \wedge (pid\_PING) \wedge (mode\_hs) \Rightarrow \neg(token = ACK))$. If the machine is in the IDLE state and high speed mode, if a stall is forced, then a PING token is ignored (or an acknowledgement is not sent out.) (Main Protocol State Machine)
- **P2**: $G^{\leq k}((state = CRC) \wedge (tx\_ready) \Rightarrow (next\_state = IDLE))$. If the machine is ready to transmit in the CRC state, the IDLE state should be reached. (Transmit/Encode State machine.)
- **P3**: $G^{\leq k}((state = CRC1) \wedge \neg(tx\_ready) \Rightarrow \neg(state = CRC2))$. The machine does not get into the CRC2 state until the device is ready to tramsmit the data. (Transmit/Encode state machine.)
- **P4**: $G^{\leq k}((state = OUT) \wedge (abort) \Rightarrow (next\_state = IDLE))$. In any OUT state, if the *abort* signal is asserted, the machine gets into IDLE mode. (Main Protocol State machine)
- **P5**: $G^{\leq k}((tokenout) \wedge (buf0\_na) \wedge (buf1\_na) \Rightarrow (signal = NACK))$. If the OUT token is received and both buffers are not available, the NACK handshake is issued. (Main Protocol State Machine)
- **P6**: $G^{\leq k}(\neg(wb\_req) \Rightarrow (state = IDLE))$. If a writeback request is not received, the machine remains in the IDLE state. (Interface State Machine.)
- **P7**: $G^{\leq k}((crc5err) \vee \neg(match) \Rightarrow (state = IDLE) \wedge \neg(send\_token))$. If an packet with bad CRC5 is received, or if there is an endpoint field mismatch in the IDLE state, then the token is ignored. (Main Protocol State Machine)

Property Checked	BMC Original	BMC Static Sliced	BMC Conditioned Slicing	Proof Result
P1	80.94	34.35	11.28	Unsat
P2	95.72	47.92	5.28	Unsat
P3	46.82	45.47	5.29	Unsat
P4	165.41	140.47	36.07	Unsat
P5	160.68	117.10	37.27	Unsat
P6	189.24	190.59	5.11	Unsat
P7	148.52	56.14	11.06	Unsat
P8	20.11	19.98	5.18	Unsat
P9	130.82	128.10	9.69	Unsat

Table 1
Comparison of execution times (in seconds) taken for verification with and without conditioned slicing. A bound of 24 was given to BMC for all experiments.

- **P8**: $G^{\leq k}(\neg(suspend\_clr) \Rightarrow \neg(state = RESUME) \wedge \neg(state = RESUME\_REQUEST))$. If the *suspend* bit is not cleared, then the machine is not resuming from the SUSPEND state. (Main State Machine)
- **P9**: $G^{\leq k}((state = SUSPEND) \wedge (T1\_gt\_2.5\mu s) \wedge (se0\_long) \Rightarrow (next\_state = RESET))$. If the machine is in the SUSPEND state and *se0_long* is asserted and there is a time lapse of 2.5μs, then the machine goes into the RESET state. (Main Protocol State Machine)

All experiments were performed using a 450 MHz UltraSparc II dual processor with 1 GB RAM.

Since the USB is a large design with several thousands of latches, model checking this design using SMV alone resulted in memory overflow.

Hence, in order to provide a baseline for our technique, we use Bounded Model Checking with a uniform bound of 24. The results were generated using the SAT-based BMC utility of the Cadence-SMV tool.

The results of our experiments are presented in Table 1. The first and second columns provide the execution times of BMC, and BMC with static slicing applied to it, respectively.

In the third column, we show the performance increase due to the application of conditioned slicing. We can observe that the performance increase

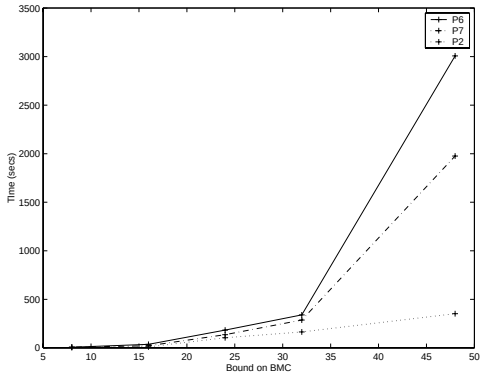

Fig. 7. Graph showing the performance gain of conditioned slicing for increasing bounds on BMC

due to static slicing is not very high as compared to BMC. In contrast, there is a tremendous gain in performance due to conditioned slicing of the design, when compared to the original as well as the statically sliced design.

We will discuss the import of these results in the next section.

Figure 7 shows the performance of some sample properties after applying our algorithm for increasing bounds of BMC. We find that there is a spectacular increase in performance due to conditioned slicing of the RTL. We placed a reasonable bound of 24, on the number clock cycles we verified. Communication protocol designs, however, need to be verified for much higher bounds. As the bound increases, the performance gain scales, too.

Properties $P1$ to $P9$ are safety properties of the form $G^{\leq k}(antecedent \implies consequent)$. Since no counterexamples to these properties are generated within the given bound, these properties are partially verified. We increased the bounds on these properties with significant performance gains.

Our algorithm performs well even on properties that do not hold and produce a counterexample. We do not reproduce them here, since they require different bounds each time.

8 Discussion

An important issue we would like to address is the improvement of conditioned slicing over static slicing. Static slicing has been applied to HDL verification before, with performance speedups. However, theoretically, static program slicing has not been shown to be different from the *cone of influence reduction* (COI) used by existing model checkers [13]. Clarke et al. [6], while comparing their static slicing technique to COI reductions, mention that COI reduction is similar to building a dependence graph for the program, and then deciphering

relevant variables using graph reachability. The dependence graph may be constructed on the HDL source code (slicing), or on the synthesized netlist. This shows that the only difference between static slicing and COI reductions, is their application domain (pre or post encoding). Semantically, the two are not different. Any performance gains due to static slicing, therefore, are due to the ease of *model generation*, as opposed to that of *model verification*.

In contrast, conditioned slicing creates a different program (or design) from static slicing or COI reductions. The Antecedent Conditioned Slice forms a new entity that does not bear similarity in structure or meaning to the original program, but retains the behavior with respect to the property in question. The performance gains due to conditioned slicing, therefore, are due to the powerful abstractions created by this technique. Although when combined with static slicing, the overall performance gain may include the model generation component, the tremendous gains in performance are primarily due to the reduction in the complexity of model verification.

9 Conclusions

The criteria for evaluating a verification methodology are *correctness, automation, precision, scaling and performance efficiency*.

Our proposed methodology has been shown to maintain correctness. It lends itself easily to automation, especially to be built on existing model checkers. Our abstractions, Antecedent Conditioned Slices, are exact, and therefore do not produce spurious counterexamples and false negatives. The experimental results show that the technique scales very well, and produces exponential performance speedups when compared to state-of-the-art techniques. The technique, therefore seems to be very promising. The technique can be applied to different domains of hardware and software verification, like microprocessor and device driver verification. Future work would focus on these varied verification application domains, where state-of-the-art model checkers are not very efficient. We also plan to extend the technique to verify liveness properties.

References

[1] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40:595–607, 1998.

[2] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. Decomposing legacy programs: a first step towards migrating to client-server platforms. *Journal of Systems and Software*, 54(2):99–110, 2000.

[3] G. Canfora, A. Cimitile, A. De Lucia, and G. Di Lucca. Software salvaging based on conditions. pages 424–433, 1994.

[4] G. Canfora, A. De Lucia, and M. C. Munro. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, 1998.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[6] E .M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[7] USB Source Code. http://www.opencores.org/pdownloads.cgi/list/usb.

[8] USB Specification Document. http://www.usb.org/developers/docs/.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, 1988.

[11] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on VHDL descriptions and its applications. pages 132–139, 1996.

[12] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[13] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[14] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, 1985.

[15] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. page 9, 1996.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[17] K. J. Ottenstein and L.M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[18] F. Tip. *Generation of Program Analysis Tools*. PhD thesis, 1995.

[19] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[20] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing: Theory and Applications*, 19(2):149–160, 2003.

[21] V. M. Vedula, W. J. Townsend, and J. A. Abraham. Program slicing for atpg-based property checking. *International Conference on VLSI Design*, pages 591–596, 2004.

[22] G. A. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 26(6):107–119, 1991.

[23] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979.

[24] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.